Software Design Document

**Victoria State Accident Data Analysis and Visualization Tool**

**Name: Sunwoo Nam**

StudentID: s5305407

Git\_log name: *nswhelen*<127074703+nswhelen@users.noreply.github.com>

**Name: Yunshuo Zhang**

StudentIDs5319113

Git\_log name: *WalnutE52* <eason1252@hotmail.com>

**Name: Ze Zeng**

StudentID: s5316065

Git\_log name: *Ze Zeng <zengze@ZeZeng.modem>*

Table of Contents

1. System Vision 3
   1. Problem Background 3
   2. System Overview 3
   3. Potential Benefits 3
2. Requirements 5
   1. User Requirements 5
   2. Software Requirements 6
   3. Use Cases & Use Case Diagrams 7

2.3.1. Use Cases: 7

2.3.2. Use Case Diagram: 8

1. Software Design and System Components 9
   1. Software Design 9
   2. System Components 10
      1. Functions 10
      2. Data Structures / Data Sources 12
      3. Detailed Design 13
2. User Interface Design 18
   1. Structural Design 18
   2. Visual Design 20

# System Vision

## Problem Background

With the rapid urbanization and growth of Victoria State, the road network has seen an increase in vehicular traffic. This growth, while indicative of economic and infrastructural development, has also led to a rise in traffic accidents. Local authorities, urban planners, and various stakeholders are in dire need of a tool that can help them understand the patterns, causes, and locations of these accidents. The Victoria State Accident Dataset, while comprehensive, is vast and complex, making it challenging to derive actionable insights without specialized tools.

## System Overview

The proposed software is a data analysis and visualization platform tailored specifically for the Victoria State Accident Dataset. It is designed to simplify the process of data interpretation, allowing users to perform a variety of analysis tasks, visualize data trends, and extract meaningful insights. The software offers an intuitive graphical interface, robust filtering tools, a range of visualization options, and predictive analysis capabilities. It aims to bridge the gap between raw data and actionable insights, providing stakeholders with the information they need to make informed decisions.

## Potential Benefits

Potential Benefits

**Enhanced Understanding**: The software aids in visualizing data, making patterns and trends more discernible, which can lead to a deeper understanding of accident causes and prevalent areas.

**Data-Driven Decisions**: Local authorities and urban planners can use the insights derived from the software to make informed decisions, potentially improving road safety.

**Informed Decision Making**: By visualizing accident hotspots and trends, local authorities can prioritize areas for infrastructure improvements, enhancing road safety.

**Efficient Resource Allocation**: With clear insights into accident patterns, resources like emergency services, road maintenance crews, and safety campaigns can be directed where they are needed the most.

**Proactive Planning**: The software's predictive analysis features allow urban planners and traffic engineers to anticipate future challenges and plan accordingly, potentially preventing accidents before they happen.

**Public Awareness**: By understanding the primary causes and locations of accidents, targeted public awareness campaigns can be launched, educating the public and promoting safer driving habits.

# Requirements

## User Requirements

**Traffic Engineers' Perspective**:

Traffic Engineers are at the forefront of ensuring the safety and efficiency of road networks. Their roles encompass the planning, design, and management of traffic systems, and they often collaborate with urban planners, civil engineers, and local authorities. Given the critical nature of their work, the software tailored to the Victoria State Accident Dataset should cater to their specific needs.

* **Intuitive Data Access**: Traffic Engineers should be able to effortlessly upload and access the dataset in its CSV format. The software should provide tools that allow them to easily navigate through the extensive data.
* **Focused Analysis Tools**: Given the vastness of the dataset, the software should offer robust filtering tools. Engineers should be able to narrow down data based on specific regions, types of accidents, or time frames, such as peak traffic hours.
* **Comprehensive Visualization**: The software must provide a range of visualization tools. Heatmaps to quickly identify accident hotspots, line graphs to track trends over time, and bar charts to compare accident frequencies across different regions or criteria are essential.
* **Predictive Capabilities**: As proactive planners, Traffic Engineers would benefit from any predictive features the software might offer. Being able to forecast potential future accident hotspots or trends based on current data would be invaluable.
* **Export and Reporting**: After their analysis, Traffic Engineers often need to present findings to various stakeholders, including city councils, planning committees, or the general public. The software should facilitate the easy export of visualizations, data summaries, and reports in commonly used formats.

## Software Requirements

* R1.1 The software shall provide an intuitive graphical user interface tailored for Traffic Engineers, ensuring ease of navigation and interaction.
* R1.2 The software shall support the direct uploading of the Victoria State Accident Dataset in its CSV format, ensuring compatibility with the specific data structure and columns.
* R1.3 The software shall offer robust data filtering tools, enabling Traffic Engineers to refine the dataset based on criteria such as specific regions, accident types, date ranges, or time frames.
* R1.4 The software shall provide a comprehensive suite of visualization tools, including heatmaps for accident hotspots, line graphs for trend analysis, and bar charts for comparative studies.
* R1.5 The software shall integrate predictive analysis capabilities, allowing Traffic Engineers to forecast potential future accident trends or hotspots based on the current dataset.
* R1.6 The software shall facilitate the export of visualizations, data summaries, and reports in popular formats such as PNG, JPEG, PDF, or even CSV for further analysis.
* R1.7 The software shall ensure data security and integrity, especially when multiple users are accessing or editing the dataset simultaneously.
* R1.8 The software shall provide a user manual or help section, offering guidance on its features, ensuring that Traffic Engineers can maximize the tool's capabilities.
* R1.9 The software shall be optimized for performance, ensuring quick data processing, visualization rendering, and minimal lag, given the potential size and complexity of the dataset.

## Use Cases & Use Case Diagrams

## 2.3.1. Use Cases:

* + 1. **Use Case: Dataset Upload and Initialization**

**Description**: Traffic Engineers need to start their analysis by uploading the dataset.

**Steps**:

1. The user accesses the upload module.
2. The software prompts for the dataset file in CSV format.
3. The user selects the dataset, and the software validates and processes it.
4. Upon successful upload, the software initializes the dataset for analysis.
   * 1. **Use Case: Focused Data Filtering**

**Description**: Given the vastness of the dataset, engineers often need to narrow down their focus.

**Steps**:

1. The user interacts with the data filtering tools.
2. The software provides options to refine data based on regions, accident types, dates, etc.
3. The user sets desired filters, and the software updates the view accordingly.
   * 1. **Use Case: Visualization Creation and Analysis**

**Description**: Visual representations are crucial for Traffic Engineers to identify patterns and trends.

**Steps**:

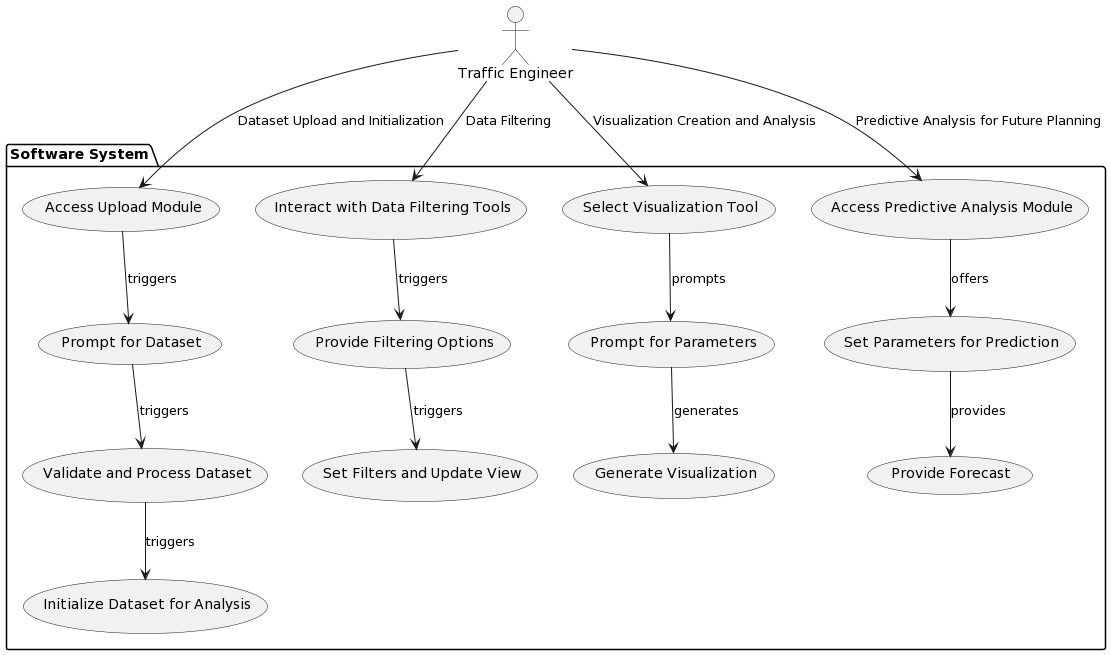
1. The user selects the desired visualization tool (e.g., heatmap, line graph).
2. The software prompts the user for any necessary parameters or settings.
3. Upon confirmation, the software generates the visualization based on the selected data.
   * 1. **Use Case: Predictive Analysis for Future Planning**

**Description**: Engineers might want to forecast future accident trends.

**Steps**:

1. The user accesses the predictive analysis module.
2. The software offers options to set parameters for the prediction.
3. The user confirms, and the software provides a forecast based on historical data and trends.

## 2.3.2. Use Case Diagram:



# Software Design and System Components

## pasted-image.tiffSoftware Design

## System Components

### Functions

**Function list:**  
**1. load\_data (filename: str) -> bool:**  
Description: Handles the dataset upload, initialization process and update.  
Input Parameters:   
 filename (string) - The CSV filename that is going be uploaded.  
Side Effects: Initializes the dataset for analysis.  
Return Value: Boolean indicating successful upload and initialization.

**2. search (dataset: list, keyword: str) -> list:**  
Description: Filters a dataset by applying a keyword-based filter to it.  
Input Parameters:   
 dataset (list) - The dataset to filter.  
 keyword (string) - The keyword to filter by.  
Side Effects: None  
Return Value: List containing the filtered results based on the keyword.

**3. apply\_filters(dataset: list, filter\_type: str, parameters: dict) -> list:**  
Description: This is the main filter function to handle all the possible filters with at most two variables.  
Input Parameters:   
 dataset (list) - The dataset to filter.  
 filter\_type (string) - Specify the filter user want to apply, such as date period, accident type.  
 parameters (string) - Specify the parameters needed of the filter, such as start\_date: ’12-05-2023’, accident type: ’Collision with vehicle’.  
 Side Effects: Updates the view with filtered data  
Return Value: List containing the filtered results based on the selected filter type.

**4. date\_range\_filter (dataset: list, start\_date: str, end\_date: str) -> list:**  
Description: Applies a date range filter to the dataset  
Input Parameters:   
 dataset (list) - The dataset to filter.  
 start\_date (string) - Start date of the range  
 end\_date (string) - End date of the range  
Side Effects: None  
Return Value: List containing the filtered results based on the date range.

**5. accident\_type\_filter (dataset: list, accident\_type: str) -> list:**  
Description: Applies a date range filter to the dataset  
Input Parameters:   
 dataset (list) - The dataset to filter.  
 accident\_type (string) - The accident type to filter by  
Side Effects: None  
Return Value: List containing the filtered results based on the accident type.

**6. create\_visualization(dataset: list, visualization\_type: str, parameters: dict)-> visualisation:**  
Description: Applies a date range filter to the dataset  
Input Parameters:   
 dataset (list) - The dataset to filter.  
 visualization\_type (string) - Type of visualization  
 parameters (dictionary) - Parameters for visualization generation  
Side Effects: Potentially utilizes member variables to store intermediate visualization data.  
Return Value: The data visualisation.

**7. export\_visualization (visualization, format: str, destination: str) - > bool**  
Description: Exports visualization in the specified format to the destination. The visualization can be a table or a chart  
Input Parameters:   
 visualization (object or data) - The visualization object or data.  
 format (string) - Desired export format,  
 destination (string) - Export location  
Side Effects: Writes visualization data to the specified export destination  
Return Value: Boolean indicating successful export.

**8. perform\_prediction (dataset: list, parameters: dict) - > visualisation or data**  
Description: Conducts predictive analysis to forecast future based on the user-defined parameters  
Input Parameters:   
 dataset (list) - The dataset to filter.  
 parameters (dictionary) - Parameters for prediction model  
Side Effects: None  
Return Value: Forecast data or insights.

### Data Structures / Data Sources

**1. List:**  
Type of Structure: Array or DataFrame  
Description: Lists are used to store the main dataset and various filtered subsets of data  
Data Members:  
 main\_dataset: Stores the entire uploaded dataset.  
 current\_dataset: Stores the filtered dataset that is currently using  
Functions: Almost all the functions in the software interact with the main dataset or subsets of data using lists

**2. Dictionary:**  
Type of Structure: Dictionary  
Description: Dictionaries are used to store user-defined filter types, visualization types and parameters for various functionalities  
Data Members:  
 filter\_parameters: Contains parameters for creating different filters  
 visualization\_parameters: Contains parameters for creating different visualizations  
Functions: create\_visualization (), apply\_filters()

**3. Visualization Object:**  
Type of Structure: Custom (Self-defined object)  
Description: Represents visualizations generated by the software. It will store all the relevant information that creating a visualization needed, such as chart type, parameters.  
Data Members:  
 Depends on the specific visualization library and approach used  
Functions: create\_visualization (), export\_visualization(), perform\_prediction()

### Detailed Design

**1. load\_data (filename: str) -> bool:**

function loadData(filename):

if filename is not ‘’:

dataset = read\_csv\_file(filename)

if dataset is not empty:

initialize\_global\_variables(dataset)

return true

else:

print("The CSV file is empty.")

return false

else:

print (“The filename is none")

return false

**2. search (dataset: list, keyword: str) -> list:**

function search (dataset, keyword):

filtered\_results = create\_empty\_list()

for each entry in dataset:

if entry\_contains\_keyword(entry, keyword):

append entry to filtered\_results

return filtered\_results

function entry\_contains\_keyword(entry, keyword):

for each field in entry:

lowercase\_field = convert\_to\_lowercase(field)

lowercase\_keyword = convert\_to\_lowercase(keyword)

If lowercase\_keyword is substring of lowercase\_field

return true

return false

**3. apply\_filters(dataset: list, filter\_type: str, parameters: dict) -> list:**

function apply\_filters (dataset, filter\_type, parameters):

if filter\_type == "date\_range":

return date\_range\_filter (dataset, parameters["start\_date"], parameters["end\_date"])

else if filter\_type == "accident\_type":

return accident\_type\_filter (dataset, parameters["accident\_type"])

else:

# Handle other filter types

return dataset

**4. date\_range\_filter (dataset: list, start\_date: str, end\_date: str) -> list:**

function date\_range\_filter (dataset, start\_date, end\_date):

filtered\_results = create\_empty\_list()

for each entry in dataset:

if start\_date <= entry\_date <= end\_date:

append entry to filtered\_results

return filtered\_results

**5. accident\_type\_filter (dataset: list, accident\_type: str) -> list:**

function date\_range\_filter (dataset, accident\_type):

filtered\_results = create\_empty\_list()

for each entry in dataset:

if entry\_date.accident\_type == accident\_type:

append entry to filtered\_results

return filtered\_results

**6. create\_visualization (dataset: list, visualization\_type: str, parameters: dict)-> visualisation:**

function create\_visualization (dataset, visualization\_type, parameters):

if visualization\_type == "bar\_chart":

data = preprocess\_data\_for\_bar\_chart(dataset)

visualization = generate\_bar\_chart(data, parameters)

else if visualization\_type == "line\_chart":

data = preprocess\_data\_for\_line\_chart(dataset)

visualization = generate\_line\_chart(data, parameters)

else:

# Handle other visualization types

visualization = None

return visualization

**7. export\_visualization (visualization, format: str, destination: str) - > bool**

function export\_visualization (visualization, format, destination):

if format == "png":

save\_as\_png(visualization, destination)

return True

else if format == "pdf":

save\_as\_pdf(visualization, destination)

return True

else:

return False

**8. perform\_prediction (dataset: list, parameters: dict) - > visualisation or data**

function perform\_prediction (dataset, parameters):

X, y = preprocess\_data(dataset)

model = initialize\_prediction\_model(parameters)

model.train(X, y)

forecast\_data = model.predict\_future(parameters["future\_dates"])

return forecast\_data

# User Interface Design

In the initial stage of designing user interfaces for software, we used wxFormBuilder, a visual interface design tool that helps create graphical user interfaces for desktop applications. Using this tool, we created a preliminary design that captured the structure and visual elements of the software interface.

Our design process is guided by user research and key findings of software requirements. By synthesizing these insights, we aim to create an intuitive and effective user interface that includes two screens: the main screen for data search and filtering, viewing data, and constructing data visualization, while the secondary screen is mainly used to construct prediction models. This meets the needs of users, enabling them to efficiently browse and analyze accident data.

## Structural Design

The interface design focuses on facilitating data exploration, visualization, and predictive modelling in a user-friendly manner. We've reorganized and refined the structural elements for enhanced usability and clarity:

**Navigation and Information Structure:**

**Main Page:**

The main page serves as the central hub for users to perform various tasks. Users can load the Victoria accident data, search and filter, and construct data visualizations. The "Make Prediction Model" button provides a direct link to the predictive modelling interface.

**Predictive Modelling Interface (Secondary Screen):**

Accessible through the main page, this interface enables users to build predictive models. Users can work with filtered data from the main page or load the original dataset. The "Export" option allows users to save their predictive models.

**Composition and Grouping:**

**Main Page:**

The main page is divided into two main sections: "Filter Data" and "Build Data Visualization."

1. “Load Data" defaults to the original dataset, while "Export Report" exports the displayed data and charts.

2. Users can filter data by searching for keywords, selecting accident types, and setting time periods

3. Filtered data is presented in the "Filtered Data" section.

4. The "Visualization" section lets users select chart types and parameters for generating charts.

**Predictive Modeling Interface (Secondary Screen):**

This interface showcases the predictive modelling process.

1. It automatically loads previously filtered data but allows users to load original data or re-filter.

2. The "Filtered Data" section displays the dataset in use.

3. Users can set titles and parameters to build prediction models.

## Visual Design

**Layout**:

The layout is structured to optimize usability and efficiency. Key elements are strategically placed to ensure easy access and navigation. The main page is divided into distinct sections for "Filter Data" and "Build Data Visualization," allowing users to focus on specific tasks. The predictive modeling interface maintains a similar structure for consistency.

**Visual Elements and Icons:**

1. Navigation Menu: Easily identifiable icons for functionalities like "Load Data," "Filter Data," "Build Visualization," and "Predictive Modeling" help users quickly grasp the available actions.

2. Filter Options: Dropdown menus and input fields are used for setting filtering parameters, ensuring familiarity and ease of interaction.

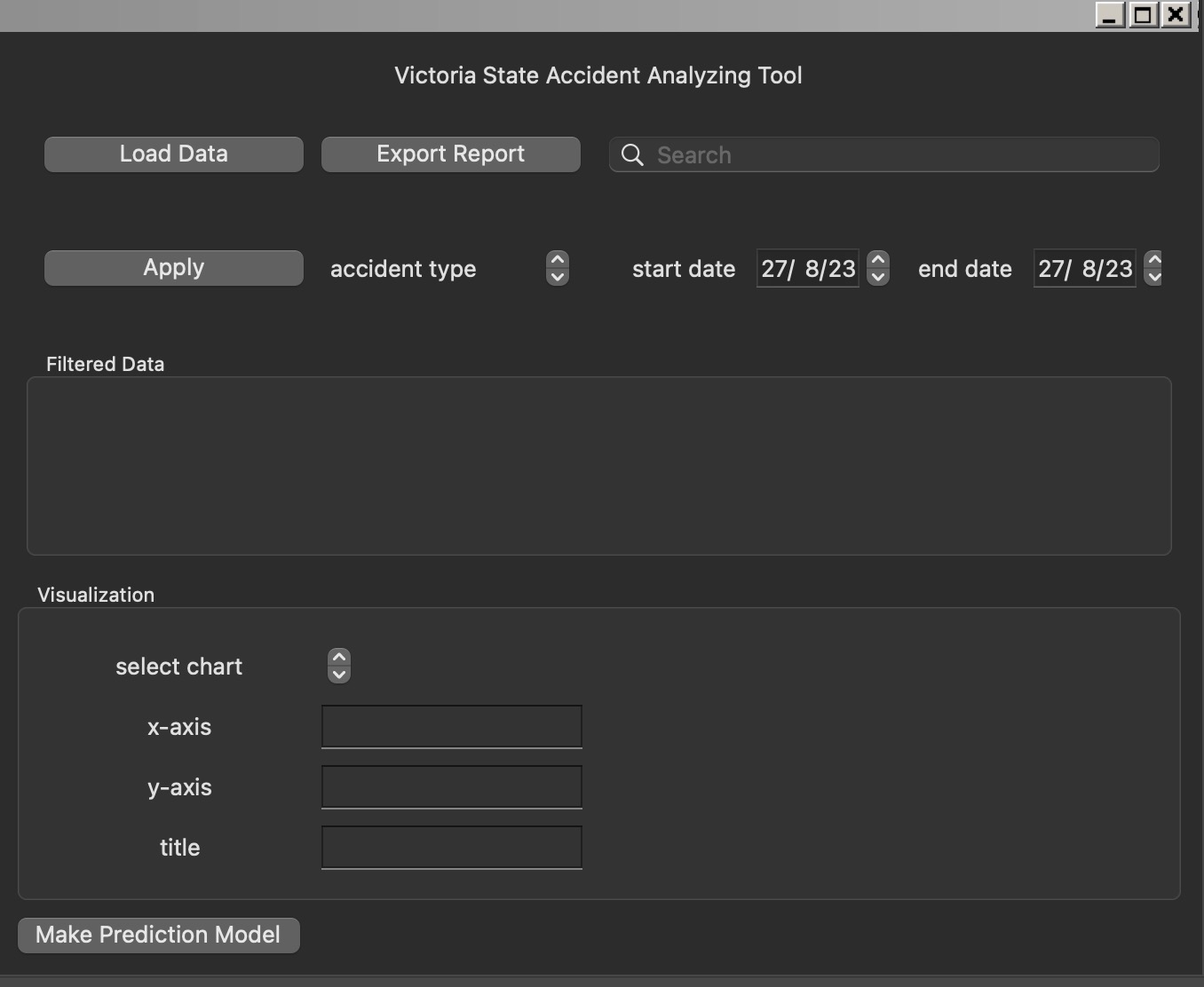
3. Visualization and Prediction Model Settings: drop-down menu facilitate users in selecting chart types. The specific input boxes help user to set parameters

4. Export and Save: Clear icons represent actions like "Export Report" reducing cognitive load and promoting action.

**Graphics and Style:**

1. Consistency: A consistent visual style is maintained across the interface, ensuring a cohesive and professional look.
2. Currently, it is the prototype of the user interface, with a focus on designing the location and functionality of each component

**Wireframe**

Main Screen:

Secondary Screen:

